

REMARKS

In this Office Action, the Examiner objected to the Drawings because the pitch of the characters is too small. Claims 1 – 18 were rejected under 35 U.S.C. §101 as being directed to non-statutory subject matter. Claims 1, 2, 4 and 5 were rejected under 35 U.S.C. §102(b) as being anticipated by Symbolic Debugging of Optimized Code by John Hennessy, ACM 1982, referred to by the Examiner as well as throughout the rest of this paper as ACM. Claim 3 was rejected under 35 U.S.C. §103(a) as being unpatentable over ACM in view of Regular Expression Pattern Matching for XML by Haruo Hosoya et al., referred to by the Examiner as well as throughout the rest of this paper as XML. Claims 5 – 18 were rejected under 35 U.S.C. §103(a) as being unpatentable over ACM in view of How Debuggers Work by J. B. Rosenberg, referred to by the Examiner as well as throughout the rest of this paper as Debug.

In response to the objection of the Drawings, attached hereto please find replacement figures to all the original figures in the Application. Applicants believe that the pitch or size of the characters in the replacement figures falls under 37 CFR §1.84(p)(3) and respectfully request withdrawal of the objection.

Claims 1 – 5 are canceled due to the 102 rejection made thereto and consequently their 101 rejection becomes moot. Claims 12 – 16 are canceled due to the 101 rejection made thereto.

Independent Claims 6, 10, 11, 17 and 18 are amended to overcome the 101 rejection made thereto. Specifically, the claims now produce a “variable” whose stored values or an “indication” when pushed on a stack is used by a user to debug a program. Applicants believe that by this amendment, the claims recite a useful, practical application and therefore fall under statutory subject matter. Hence, withdrawal of the 101 rejection, as applied to Claims 6 – 11, 17 and 18, is kindly requested.

New Claims 19 – 24 are provided for consideration. No new matter is added since support for the new claims can be found from the second full paragraph on page 12 to the second full paragraph on page 17.

CA920030042US1

By this Amendment therefore, Claims 6 – 11 and 17 - 24 are pending in the Application. For the reasons stated more fully below, Applicants submit that the pending claims are allowable over the applied references. Hence, reconsideration, allowance and passage to issue are respectfully requested.

As disclosed in the SPECIFICATION, debugging code often requires that breakpoints be placed in the code by the developer so that, when the code is compiled and executed, the execution pauses at the breakpoint. While execution of the code is paused, the developer may, for instance, review the values of certain variables or determine that a lock has been obtained on a given object. Once the developer has reviewed the available information for the breakpoint, a command may be given so that execution may continue, at least until execution pauses at the next breakpoint.

When lines of program code prepared by a software developer are executed in a runtime, the code may be optimized by the compiler so that, when executed, the program runs more efficiently than would have been otherwise. Compilers that optimize code may scan through the code and use a set of use logical rules to determine where efficiencies may be found through the elimination of various instructions and, often, the introduction of some new instructions to compensate for some of the eliminated instructions.

When a program flow is optimized, nodes and connections in that flow may be repositioned and reorganized such that the flow is more efficient. Consequently, it may be said that there are two types of flow: a user defined flow, which is developed in the tooling (the writing environment); and an optimized flow, which is used in the runtime (the execution environment). Because of this rearrangement, a flow debugger cannot always identify a connection in the optimized flow that corresponds to a given connection in the user defined flow. Hence, where the user (software developer) has placed a breakpoint on the given connection between nodes in the user defined flow, this breakpoint may not map well to a particular connection in the optimized flow. As such, it may not be clear to the debugger

precisely where to place corresponding breakpoint in the optimized flow, i.e., where to pause execution.

One solution to this problem is to avoid optimization while debugging. That is, rather than producing an optimized flow, the user defined flow is compiled to produces a special "debuggable" version of the user defined flow for execution. The developer may then use the results of the execution of the debuggable version of the flow to find errors and inconsistencies. However, by doing so, the developer is not debugging a "true" (optimized) version of the executable code that will be run in a finished product.

Through the use of mapping of flow connections and maintenance of information about connections on which breakpoints have been placed and acted upon, an optimized flow corresponding to a user defined flow may be executed for debugging. The connection information maintenance may be seen to allow for a debugger user interface that is notable for clarity and minimization of confusion of the user.

The present invention provides a way of debugging a program code that provides such clarity and minimization of confusion. As in previous cases, the program code to be debugged will include at least one node that will have a breakpoint that is to be used to look at the value of a variable during execution of the program. This value of the variable is what is used to debug the program.

In any case, the program code will then be optimized. The optimized code may have at least two optimized nodes that are derived from the one node of the program code. If the optimized code has at least these two optimized nodes, during execution of the optimized code, it will be ascertained that two values of the variable are stored, one value for each optimized node traversed. Since each time an optimized node is traversed a value of the variable is stored, it will be clear to the user which value of the variable is attributed to which one of the two optimized nodes traversed as opposed to when only one value of the variable is stored for the two nodes.

CA920030042US1

The invention is set forth in claims of varying scopes of which Claim 19 is illustrative.

19. A method of debugging a program code, the program code including at least one node having a breakpoint and a variable, the breakpoint for allowing a user to use a value of the variable to debug the program code, the method comprising:

generating an optimized flow from the program code, the generated optimized flow having at least two optimized nodes derived from the one node having the breakpoint;

executing the optimized code;

ensuring that two values of the variable are stored during execution of the optimized code, each one of the values being stored when one of the at least two optimized nodes is traversed; and

debugging the program code using the stored values of the variable. (Emphasis added.)

Pending Claims 6 – 11, 17 and 18 were rejected using ACM in combination with Debug. Applicants did not receive a copy of Debug and thus cannot assess the relevance of Debug to the pending claims. Applicants kindly request a copy of the reference.

Nonetheless, note that if ACM does not teach the use of stacks, stack operations and internals of stack operations as stated by the Examiner, there is no reason for ACM to teach the steps of receiving an instruction to push an indication of a particular user flow connection, among said at least one user flow connection associated with said given optimized flow connection, into said stack; and responsive to receiving said instruction to store, pushing said indication of said particular user flow connection into said stack.

Regarding the newly added claims (i.e., Claims 19 – 24), Applicants submit that they are allowable over ACM and XML.

ACM purports to teach a symbolic debugging of optimized code. A symbolic debugger, as described by ACM, is a program development tool that interacts with a

CA920030042US1

running program and a programmer at the source program level. The symbolic debugger can stop the program and examine any currently active variable.

Program optimization, by its very nature, alters the structure and intermediate result of the program. Consequently, if an optimized program aborts or a debugger stops the optimized program, the resulting values of the variables may not equal the values of those variables at the corresponding point in an execution of the original source program. Hence, a programmer will have to unravel the optimized code to determine what values the variables would have had if the source code was executing instead of the optimized code.

ACM then provides a manner by which a programmer may more easily determine the values that the variables should have when an optimized program aborts or a debugger stops the optimized program. Specifically, ACM divides optimization into local and global code optimizations and provides a manner by which the values of the variables may be gleaned when using either one of the optimizations.

However, ACM does not teach, show or so much as suggest the steps of ***ensuring that two values of the variable are stored during execution of the optimized code, each one of the values being stored when one of the at least two optimized nodes is traversed; and debugging the program code using the stored values of the variable*** as claimed.

XML purports to teach a regular expression pattern matching for XML. Regular expression patterns are intended to be used in languages whose type systems are also based on the regular expressions. To avoid excessive type annotations, XML develops a type inference scheme that propagates type constraints to pattern variables from the surrounding contexts.

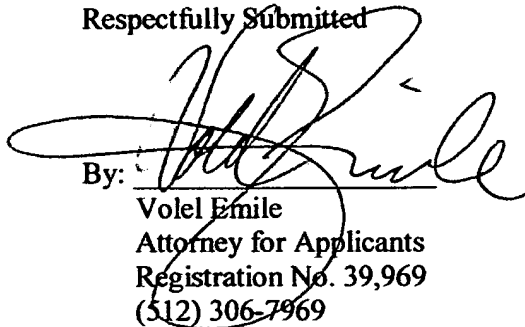
However, just as in the case of ACM, XML does not teach, show or so much as suggest the steps of ***ensuring that two values of the variable are stored during execution of the optimized code, each one of the values being stored***

when one of the at least two optimized nodes is traversed; and debugging the program code using the stored values of the variable as claimed.

As mentioned above, Applicants did not receive a copy of Debug. Therefore, Applicants are unable to assess its relevance on the added claims.

Nevertheless, since the Examiner stated that Debug is used to teach stack operations, Applicants submit that the Claim 19 – 33 are also allowable. Consequently, Applicants once more respectfully request reconsideration, allowance and passage to issue of the pending claims in the application.

Respectfully Submitted

By: 
Volel Emile
Attorney for Applicants
Registration No. 39,969
(512) 306-7969